1. **What is the role of a container orchestration tool?**
   Container orchestration tools like Kubernetes automate the deployment, scaling, and management of containerized applications.

2. **What is the role of a service mesh in microservices architecture?**
   A service mesh provides a dedicated infrastructure layer for handling service-to-service communication, offering features like load balancing, security, and observability.

3. **Explain the role of a reverse proxy in microservices architecture?**
   A reverse proxy plays a crucial role in a microservices architecture by acting as an intermediary between client requests and the individual microservices that make up the application. Its primary function is to handle incoming requests and route them to the appropriate microservice, based on factors such as URL paths, headers, or other criteria.

4. **Explain CAP theorem?**

   The CAP Theorem (also known as Brewer's Theorem, after computer scientist Eric Brewer) is a fundamental principle in distributed systems that describes the trade-offs between three core properties:

   C - Consistency
   A - Availability
   P - Partition Tolerance

   The theorem states that in a distributed system, you can only achieve at most two of these three properties simultaneously. Let's break down what each of these properties means:

   Consistency (C): Definition: Every read operation will return the most recent write (or an error), ensuring that all nodes in the system see the same data at the same time. **Explanation:** If a system is consistent, it guarantees that when a user reads from the database, they will always see the latest data, regardless of which node they query. This eliminates issues like reading stale or outdated data. **Availability (A):**Definition: Every request (read or write) will receive a response, either with the requested data or an error message (but the system will always respond). **Explanation:** If a system is available, it ensures that even if one or more nodes fail, the system continues to operate and provides a response to all requests. This might involve returning stale or partial data, but the system guarantees it won't be unresponsive. **Partition Tolerance (P):** Definition: The system will continue to function properly even if network partitions occur (i.e., some nodes cannot communicate with each other due to network failures). **Explanation:** Partition tolerance is essential in any distributed system because network partitions are inevitable in large-scale systems. The system must be resilient to communication failures and keep functioning despite them.

   Real-World Examples:

CA Systems (Consistency + Availability): Systems that prioritize consistency and availability but can't handle network partitions well. Typically used in closed environments where network partitions are unlikely.

CP Systems (Consistency + Partition Tolerance): Systems that prioritize consistency and network partition tolerance at the expense of availability. An example is a traditional relational database with strong consistency requirements.

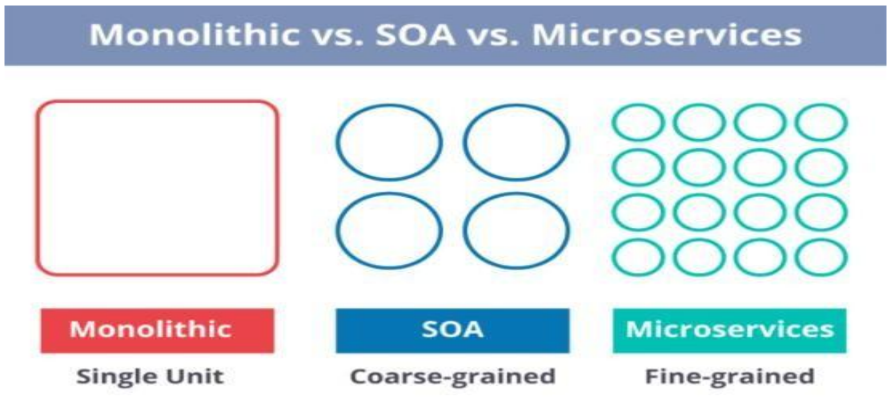
AP Systems (Availability + Partition Tolerance): Systems that prioritize availability and partition tolerance at the expense of consistency. An example is a NoSQL database like Cassandra or DynamoDB.

5. **How does the CAP theorem relate to a microservices architecture?**
   The CAP theorem states that a distributed system cannot simultaneously provide Consistency, Availability, and Partition tolerance. In microservices, architects need to make trade-offs based on these factors.

6. **How do you handle inter-service communication timeouts and retries?**
   Timeouts and retries can be managed through the use of patterns like Circuit Breaker and Retry, which help prevent cascading failures and ensure more reliable communication.

7. **Explain blue-green deployment in microservices?**
   Blue-green deployment in microservices involves running two identical environments - one serves production traffic (blue), while the other hosts a new version (green). Traffic is gradually switched from blue to green, allowing for seamless updates with minimal downtime. If issues arise, traffic can be rerouted back to the stable blue environment, ensuring reliability and enabling quick rollback if necessary

8. **What is the difference between Monolithic, SOA and Microservices Architecture?**



Monolithic Architecture is similar to a big container wherein all the software components of an application are assembled together and tightly packaged.
A Service-Oriented Architecture is a collection of services which communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity.
Microservice Architecture is an architectural style that structures an application as a collection of small autonomous services, modelled around a business domain.

9. What are different types of Tests for Microservices?

- At the **bottom level**, we have **technology-facing tests** like- unit tests and performance tests. These are completely automated.
- At the **middle level**, we have tests for **exploratory testing** like the stress tests and usability tests.
- At the **top level,** we have **acceptance tests** that are few in number. These acceptance tests help stakeholders in understanding and verifying software features.

10. What is the use of PACT in Microservices architecture?

PACT is an open source tool to allow testing interactions between service providers and consumers in isolation against the contract made so that the reliability of Microservices integration increases.

Usage in Microservices:

- Used to implement Consumer Driven Contract in Microservices.
- Tests the consumer-driven contracts between consumer and provider of a Microservice.

11. What is a Bounded Context?

Bounded Context is a central pattern in Domain-Driven Design.

Bounded Context, the word explains itself, is part of a larger context. The context is actually the environment in which you will develop the application. And a Bounded Context is a logical part of that context.
Suppose, tomorrow you start a leasing company. Your leasing company starts with one person and you start from scratch. You have some Excel documents in which you keep track of everything. In the beginning you do everything: order processing, marketing, customercare,… You will only have one system, a folder on your computer, where you store everything.
Tomorrow you will work with ten people. Then you might already have a bit more structure. You start to split everything within your company. The idea behind Bounded Context is that you do the same in software-development too. So the intention is actually to limit the complexity, by creating separate contexts that communicate with each other.

12. Difference b/w TDD, DDD and BDD?

Test-Driven Development (TDD)
Focus: Code quality through automated testing.
Approach: Write tests before writing the actual code. This cycle involves writing a failing test, **writing** the minimum code to pass the test, and then refactoring the code. **Workflow**: Red (fail) → Green (pass) → Refactor.

Domain-Driven Design (DDD)
Focus: Capturing the domain knowledge and business logic.

Approach: Collaboratively define a model that reflects the domain. Break down complex domains into smaller, more manageable subdomains and bounded contexts. Use techniques like domain models, ubiquitous language, and aggregates.
Workflow: Domain analysis → Modeling → Implementation.

Behavior-Driven Development (BDD)
Focus: Creating business-readable specifications that guide development.
Approach: Write specifications in natural language that describe the behavior of the system. These specifications are written in the form of "Given-When-Then" scenarios that can be automated as tests.
Workflow: Specification → Automation → Development.

13. How Do Microservices Communicate With Each Other?

HTTP/REST: Microservices communicate through HTTP endpoints using RESTful APIs.
Messaging/event-driven: Asynchronous communication via message brokers or event-driven architectures.
RPC (remote procedure call): Invoking methods or procedures on remote services.
Message queues: Services exchange messages via message queues or brokers.
Service mesh: Infrastructure layer for managing communication, service discovery, and load balancing.
Shared database: Microservices communicate indirectly by sharing a common database.

14. What Are Some Common Microservices Patterns?

API g**ateway pattern**: Provides a single entry point for clients to access multiple microservices, handling routing, authentication, and monitoring.
Circuit breaker pattern: Handles failures and prevents cascading failures by monitoring service availability and redirecting requests to a fallback mechanism when necessary. **Saga pattern:** Manages long-running, distributed transactions by breaking them down into smaller, compensating transactions across multiple services.
Event sourcing pattern: Captures all changes to an application's state as a sequence of events, allowing for reliable audit logs and flexible querying.
Command query responsibility segregation (CQRS) Pattern: Separates read and write operations, enabling optimized read operations and handling complex data requirements.
Bulkhead pattern: Divides microservices into separate pools of resources to prevent failures in one service from impacting others.
Retry pattern: Automatically retries failed requests or operations with increasing delays to handle transient failures in communication.

15. What Are the Various Deployment Strategies Commonly Used in Microservices?

Individual service deployment: Each Microservice is deployed independently.
Containerization: Microservices are packaged as containers (e.g., Docker).
Orchestration with Kubernetes: Microservices are managed using Kubernetes. **Serverless deployment**: Microservices are deployed using serverless platforms like AWS Lambda or Azure Functions.

Blue-green deployment: Gradual shift from the old version to the new version.
Canary deployment: New version deployed to a small subset of users or traffic.
Rolling deployment: Gradual deployment of the new version across the infrastructure.
Feature toggling: Selective activation or deactivation of features within a Microservice

16. **Explain about Canary deployment?**
    In a canary deployment, the new version of a software application is rolled out to a small subset of users first rather than deploying it to the entire user base at once. This allows you to monitor the impact of the update on a limited number of users and ensure that it doesn't introduce critical issues before it is rolled out more broadly.

    How Canary Deployment Works:

    Two Versions Running Simultaneously: Both the old (stable) and new (canary) versions of the application run simultaneously, with a small percentage of traffic directed to the canary version.
    Monitoring and Evaluation: The performance and behaviour of the canary version are closely monitored. Metrics such as user engagement, error rates, and system performance are evaluated.
    Gradual Rollout: If the canary version performs well, the update is gradually rolled out to more users1. If issues are detected, the deployment can be halted, and the system can revert to the stable version.
    Final Deployment: Once the canary version is verified to be stable and issue-free, the update is deployed to all users.

17. **How Can You Achieve Fault Tolerance and Resilience in Microservices?**

    Redundancy and replication: Run multiple instances of each microservice for backup and availability.
    Circuit breaker pattern: Detect and handle service failures to prevent cascading failures.
    Bulkhead pattern: Isolate failures within limited scopes to contain their impact.
    Timeout and retry strategies: Set timeouts and implement retries for transient failures.
    Failure monitoring and alerting: Monitor system health, log errors, and set up alerts for critical failures.
    Health checks and self-healing: Implement health checks and automated recovery mechanisms.

18. **What Are the Various Deployment Strategies Commonly Used in Microservices?**

    Individual service deployment: Each Microservice is deployed independently.

    Containerization: Microservices are packaged as containers (e.g., Docker).

    Orchestration with Kubernetes: Microservices are managed using Kubernetes.

    Serverless deployment: Microservices are deployed using serverless platforms like AWS Lambda or Azure Functions.

Blue-green deployment: Gradual shift from the old version to the new version.
Canary deployment: New version deployed to a small subset of users or traffic.

Rolling deployment: Gradual deployment of the new version across the infrastructure.

Feature toggling: Selective activation or deactivation of features within a Microservice.

19. **What is Reactive Extension in Microservices?**

    Reactive Extension is a design pattern that allows collecting results by calling multiple services and then compiles a combined response. It is also called Rx. Rx is a popular tool in distributed systems that works opposite to legacy flows.

20. **What is the difference between Rest and Microservices**?

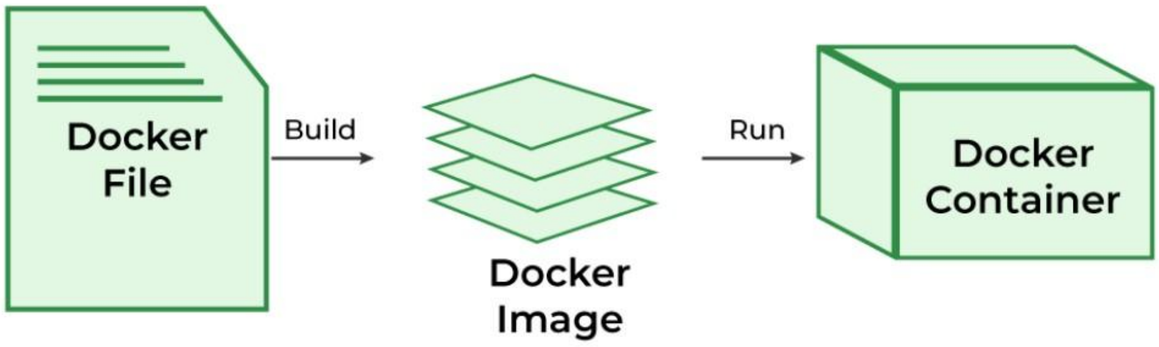
    There are multiple ways to implement microservices. REST over HTTP is one of them. REST is also used in other applications such as web apps, API design, and MVC applications to serve business data.
    On the other hand, in microservices architecture, all the system components are put into individual components, which can be built, deployed, and scaled individually. Microservices provide certain principles and best practices that help in building a resilient application. So, we can say that REST is a medium to build Microservices.

21. **Explain about Docker?**

    Docker is an open-source containerization platform by which you can pack your application and all its dependencies into a standardized unit called a container. Containers are light in weight which makes them portable and they are isolated from the underlying infrastructure and from each other container. You can run the docker image as a docker container in any machine where docker is installed without depending on the operating system.

    Key Components of Docker
    The following are the some of the key components of Docker:
    Docker Engine: It is a core part of docker, that handles the creation and management of containers.
    Docker Image: It is a read-only template that is used for creating containers, containing the application code and dependencies.
    Docker Hub: It is a cloud based repository that is used for finding and sharing the container images.
    Dockerfile: The Dockerfile uses DSL (Domain Specific Language) and contains instructions for generating a Docker image. Dockerfile will define the processes to quickly produce an image. While creating your application, you should create a Dockerfile in order since the **Docker daemon** runs all of the instructions from top to bottom.

    Docker Registry : It is a storage distribution system for docker images, where you can store the images in both public and private modes.



22. Explain about Kubernetes?

Kubernetes is an open-source Container Management tool that automates container deployment, container scaling, descaling, and container load balancing (also called a container orchestration tool). It is written in Golang and has a vast community because it was first developed by Google and later donated to CNCF (Cloud Native Computing Foundation). Kubernetes can group 'n' number of containers into one logical unit for managing and deploying them easily. It works brilliantly with all cloud vendors i.e. public, hybrid, and onpremises.

Deploying and Managing Containerized Applications With Kubernetes
Follow the steps mentioned below to deploy the application in the form of containers. **Step 1:** Install kubernetes and setup kubernetes cluster there should be minimum at least one master node and two worker nodes you can set up the kubernetes cluster in any of the cloud which are providing the kubernetes as an service.
Step 2: Know create deployment manifestfile you can create this manifests in the manifest you can specify the exact number of pods are required and what the container image and what types of resources are required after completion of writing the manifestfile apply the file using kubectl command.
Step 3: After creating the pods know you need to expose the service to the outside of the for that you need to write one more manifestfile which contains service type (e.g., LoadBalancer or ClusterIP), ports, and selectors.

23. Explain about Rate Limiting in Micro Services?

Rate limiting in microservices is a crucial technique to control the number of requests a service can handle within a specific time frame. It helps prevent system overload, ensures fair usage, and enhances security by mitigating abuse (e.g., DDoS attacks). Here's a breakdown of rate limiting in microservices:

Fixed Window: Limits requests per fixed time window (e.g., 100 requests per minute).Simple but can lead to burst traffic issues at the start of a window.
Sliding Window: More refined version of fixed window, using rolling counters. Provides smoother request handling over time.

Token Bucket: Requests are processed as long as tokens are available. Tokens regenerate over time, allowing occasional bursts.
Leaky Bucket: Requests are processed at a constant rate. Helps smooth out traffic spikes.
Adaptive Rate Limiting: Adjusts limits dynamically based on system health and demand. Uses ML or heuristics to modify limits in real-time.

Best Practices:

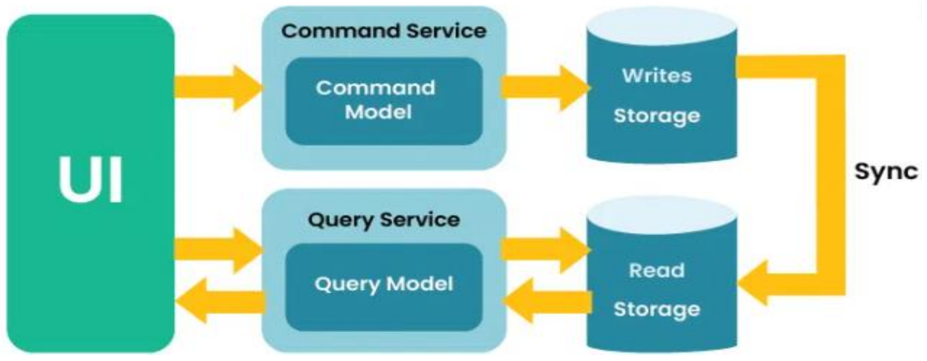
Use Distributed Caching: Redis is a common choice.
Different Limits for Different Users: VIP users may have higher limits.
Implement Retry Strategies: Inform users when throttled (HTTP 429).
Monitor & Log Rate-Limited Requests: Use Prometheus/Grafana for monitoring.
Combine with Circuit Breakers: Tools like Resilience4j help with fault tolerance.

24. Explain about CQRS?

Certainly! **CQRS**, or **Command Query Responsibility Segregation**, is a design pattern used in software engineering to separate databases for read and write operations for a data store. This approach helps to improve performance, scalability, and security by using different models to handle reading data and writing data.

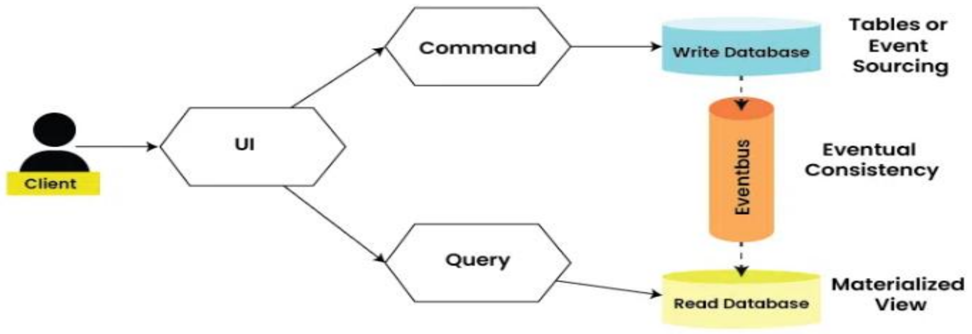
Commands: These are requests to perform an action or change the state of the application. Commands mutate the state of the system but do not return any data. Examples include creating a new user, updating an order, or deleting a product.

Queries: These are requests to retrieve data from the system. Queries do not change the state of the application and only return data. Examples include fetching user details, retrieving order history, or getting product information.



25. How to Sync Databases with CQRS Design Pattern?

Synchronizing databases in a system that follows the CQRS pattern can be challenging due to the separation of the write and read sides of the application.



- **Step 1: Write to the Command Database**: When you make changes (create, update, delete), they are first saved in the **command database**. This database is optimized for handling write operations.
- **Step 2: Generate Events**: After the write operation is successful, the system generates **events** that describe what changed (like “Order Created” or “User Updated”). These events serve as notifications about the updates.
- **Step 3: Update the Query Database**: The read database, optimized for fast queries, listens for these events and **applies the changes** to its own copy of the data. This way, the query database gets updated with the latest information.
- **Step 4: Eventual Consistency**: The key idea is that the query database doesn’t have to update immediately. There can be a slight delay, but eventually, both databases will sync, ensuring consistency over time.